1. Overview
    1. Introduction to OFDM and SDRs
2. Transmitter
    1. MATLAB
        1. Bits-to-Words (Transmitter) (MATLAB)
        2. Words-to-Symbols (Transmitter) (MATLAB)
        3. Symbols-to-FFT (Transmitter) (MATLAB)
        4. Add Cyclic Prefix (Transmitter) (MATLAB)
        5. OFDM Symbol Generator (Transmitter) (MATLAB)
    2. LabVIEW
        1. Bits-to-Words (Transmitter) (LabVIEW)
        2. Words-to-Symbols (Transmitter) (LabVIEW)
        3. Symbols-to-FFT (Transmitter) (LabVIEW)
        4. Add Cyclic Prefix (Transmitter) (LabVIEW)
        5. Windowing (Transmitter) (LabVIEW)
        6. OFDM Symbol Generator (Transmitter) (LabVIEW)
3. Receiver
    1. MATLAB
        1. Time Synchronizaton (Receiver) (MATLAB)
        2. Frequency Synchronization (Receiver) (MATLAB)
        3. FFT-to-Symbols (Receiver) (MATLAB)
        4. Symbols-to-Words (Receiver) (MATLAB)
        5. Words-to-Bits (Receiver) (MATLAB)
        6. OFDM Symbol Decoder (Receiver) (MATLAB)
    2. LabVIEW
        1. Frequency Synchronization (Receiver) (LabVIEW)
        2. FFT-to-Symbols (Receiver) (LabVIEW)
        3. Equalization (Receiver) (LabVIEW)
        4. Symbols-to-Words (Receiver) (LabVIEW)
        5. Words-to-Bits (Receiver) (LabVIEW)
        6. OFDM Symbol Decoder (Receiver) (LabVIEW)

4. Tests
    1. All Software (Tests)
5. References
    1. References

Introduction to OFDM and SDRs

What is OFDM? What is an SDR? Why is either important or relevant? This module addresses the nature of these questions as well as what the project aims to implement.

OFDM (Orthogonal Frequency Division Multiplexing) is the technique for transmitting data in parallel using large number of modulated carriers with harmonic frequency spacing so that the carriers are orthogonal to each other. The orthogonality allows spectral overlapping of channels that can be separated later, much like quadrature modulation.

SDR (Software Defined Radio) refers to a radio communication system that can be configured to send and receive a wide range of modulated digital signals across a large frequency spectrum by means of a programmable hardware platform. With the advancements made in low noise amplifiers (LNAs), analog-to-digital converters (ADCs) or samplers, and antenna technology, SDRs have become an emergent technology in communications. This will allow the same antenna or antenna array to be used for different frequency ranges, and to move the ADC as close as possible to the antenna with little or no pre-filtering, and performing the entire signal processing digitally.

In a nut shell, OFDM is a modulation scheme that rides on top of another basic type of modulation such as BPSK and QAM to allow simultaneous transmission of independent signal carriers. It is highly scalable, allowing expansion or reduction of the signal bandwidth to accommodate the dynamic creation or removal of signal carriers. As a result these unique properties, it is widely used in a variety of important applications such as mobile radio and digital broadcasting. It has also been touted as the possible scheme of choice for the predicted paradigm shift known as the cognitive radio. In our Digital Signal Processing Laboratory project, we gained a firm grasp of this exciting technology by creating an OFDM transmitter and implementing a corresponding OFDM software receiver. The transmitter modulates BPSK signals and outputs them through a communications FPGA, while the receiver uses Matlab libraries to process the captured signals to retrieve the original data. At the end of the project, the final

system was shown to be able to correctly transmit and receive 64 independent signal carriers simultaneously.

The new project, an independent study carried on by one of the three original students with Dr. Christopher Schmitz of the University of Illinois at Urbana/Champaign, will be an all LabVIEW transceiver implemented in actual hardware. It will run on a front-end receiver utilizing National Instruments' PXI chassis populated with the 5660 digital downconverter/high-speed digitizer in conjunction with the 5671 AWG/upconverter. The goal is a real-time system that transmits and receivers over the air, so that parameters can actively be tweaked, and the resulting changes in performance can be observed.

Bits-to-Words (Transmitter) (MATLAB)
The fairly trivial operation of converting a set of bits to a set of words.
NOTE: The MATLAB library was used in the initial version of the design but has since been replaced by an updated LabVIEW module. Please see the LabVIEW portion of the transmitter for the most current version.

```
%% Bits to Words function words = b2w(bits, bits_per_symbol) %% num_subcarriers = size(bits,2)/bits_per_symbol; words = zeros(num_subcarriers, bits_per_symbol); for n=1:num_subcarriers words(n,1:bits_per_symbol) = bits((n-1)*bits_per_symbol+1:(n-1)* (bits_per_symbol)+bits_per_symbol); end end
```

Words-to-Symbols (Transmitter) (MATLAB)
This code coverts a set of binary words to a set of complex symbols through the use of decimal indexing of a word-to-symbol map. **NOTE: The MATLAB library was used in the initial version of the design but has since been replaced by an updated LabVIEW module. Please see the LabVIEW portion of the transmitter for the most current version.**

```
%% Words to Symbols function symbols =
w2sym(words, word_to_symbol_map, bits_per_symbol)
%% num_subcarriers = size(words,1); symbols =
zeros(1,num_subcarriers); for n=1:num_subcarriers
symbols(1,n) =
word_to_symbol_map(binvec2dec(fliplr(words(n,
1:bits_per_symbol)))+1); end end
```

Symbols-to-FFT (Transmitter) (MATLAB)
This code converts a group of complex symbols to a single complex baseband OFDM period in the time domain. Equivalently, it can be thought of as a parallel to serial operation while simultaneously modulating all the subcarriers by their complex amplitude. **NOTE: The MATLAB library was used in the initial version of the design but has since been replaced by an updated LabVIEW module. Please see the LabVIEW portion of the transmitter for the most current version.**

```
%% Symbols to FFT function [re im] =
sym2fft(symbols, size_of_fft) %% num_subcarriers =
size(symbols,2); input = [ 0 symbols(1:
(num_subcarriers/2)) zeros(1,size_of_fft-
num_subcarriers-1)
symbols((num_subcarriers/2)+1:num_subcarriers) ];
output = ifft(input); re = real(output); im =
imag(output); end
```

Add Cyclic Prefix (Transmitter) (MATLAB)
This performs the simple but crucial task of adding the cyclic prefix for the transmission guard interval. This will protect real systems against multipath fading. **NOTE: The MATLAB library was used in the initial version of the design but has since been replaced by an updated LabVIEW module. Please see the LabVIEW portion of the transmitter for the most current version.**

```
%% Symbols to FFT function [re_cyc im_cyc] =
cyc(re, im, perc_cyc) %% size_of_fft = size(re,2);
size_of_cyc = round(size_of_fft*(1-perc_cyc))+1;
re_cyc = [re(size_of_cyc:size_of_fft) re]; im_cyc
= [im(size_of_cyc:size_of_fft) im]; end
```

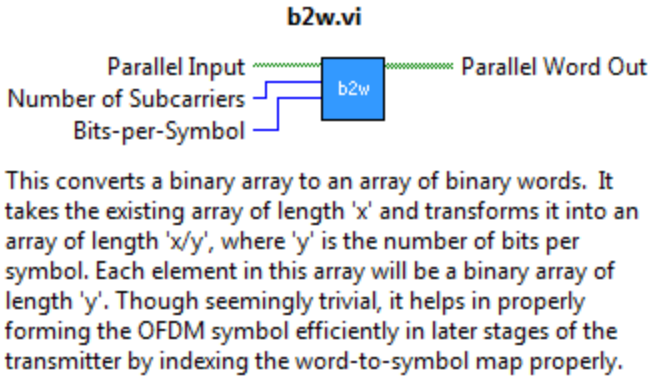
OFDM Symbol Generator (Transmitter) (MATLAB)
Puts together the previous MATLAB scripts to generate a full OFDM Symbol. **NOTE: The MATLAB library was used in the initial version of the design but has since been replaced by an updated LabVIEW module. Please see the LabVIEW portion of the transmitter for the most current version.**

```
%% Generate Baseband OFDM Signal function [re im] = ofdm_gen(bits, bits_per_symbol, word_to_symbol_map, size_of_fft, perc_cyc) %% words = b2w(bits,bits_per_symbol); symbols = w2sym(words,word_to_symbol_map,bits_per_symbol); [re_0 im_0] = sym2fft(symbols,size_of_fft); [re im] = cyc(re_0,im_0,perc_cyc); end
```

Bits-to-Words (Transmitter) (LabVIEW)

This is a seemingly trivial sub-VI that converts a parallel stream of bits to a parallel group of bit-words. While it could easily be done without a sub-VI, it's always important to practice modularity and simplicity for readability in the overall design. This makes the debugging and modification process, an inevitability, much simpler in the future.

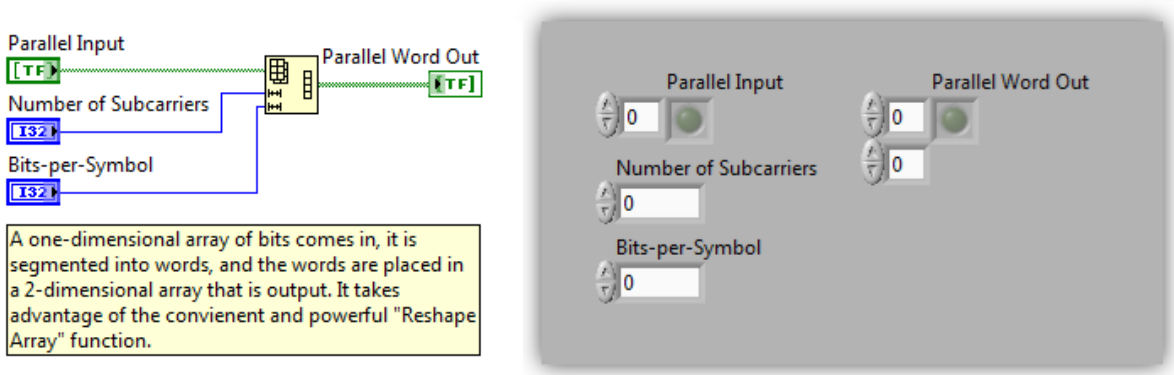
Input/Outputs and Help



This is the LabVIEW help and block description for the Bits-to-Words sub- VI.

The sub-VI for the rather trivial operation of converting a string of bits to bit-words is shown above in Figure 1. Only basic knowledge of LabVIEW is required to implement this module. While some may argue it's trivial and non-deserving of its own sub-VI, modularity and readability still prevail in good design technique, and this transmitter is no exception.

Block Diagram Layout



This is the LabVIEW block diagram for the Bits-to-Words sub-VI.

Figure 2 above shows the block diagram layout. The parallel bits are taken as an input and fed into the built-in LabVIEW function "Reshape Array," which groups the number of bits appropriately.

For the logisitics in actually constructing this function, see the video below. For any additional questions, the example usage video in Figure 4, or email the author.

Instructional Video

[missing_resource: http://www.youtube.com/v/JGGlo6h3SYA&hl=en&fs=1&rel=0]

This is the instructional video for constructing the Bits-to-Words sub-VI.

Example Video

[missing_resource: http://www.youtube.com/v/Bth8nkZ9C5A&hl=en&fs=1&rel=0]

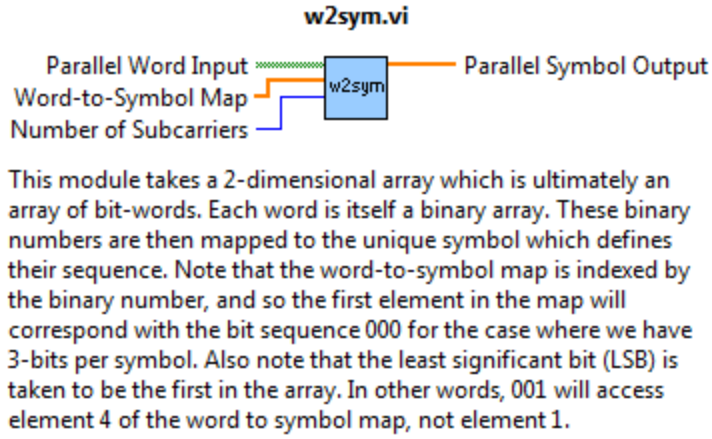
This is the example video for using the Bits-to-Words sub-VI.

Download This LabVIEW sub-VI

Words-to-Symbols (Transmitter) (LabVIEW)

This is the educational tutorial for creating a bit-word to complex symbol map. Though a simple function, it is certainly key to implementing the OFDM transmitter chain, and will become equally important when retrieving the data on the other end of the transceiver.

Input/Outputs and Help

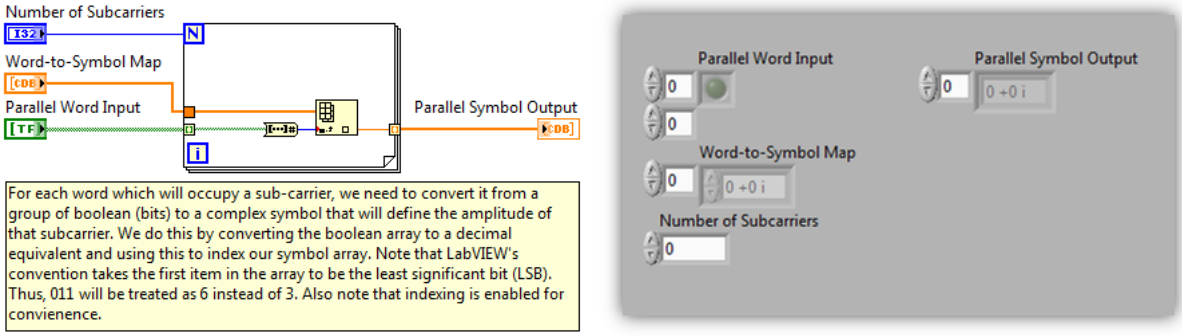


This is the LabVIEW help and block description for the Words-to-Symbols Sub-VI.

The above Figure (Figure 1) shows the Words-to-Symbols function. This is fairly straightforward: in band limited channels, we cannot send a square-wave stream of 1's and 0's. More importantly, we'd like to leverage OFDM's attractive properties and transmit many data streams on multiple carriers.

Thus we must take each word (itself a group of bits), and assign it a unique complex signal. This is where any popular scheme is chosen such as BPSK, M-ary QAM, etc. By simply defining the word-to-symbol map, the numbers the words represent in the base 10 (decimal) system index elements in the map and the conversion is done.

Block Diagram Layout



This is the LabVIEW block diagram for the Words-to-Symbols sub-VI.

Figure 2 shows this simple process. As an example, let's say the bit-word is 110. LabVIEW's convention would read this as decimal 3, as the first item in the array is seen as the least significant bit (LSB). Next, we index our symbol array with decimal 3. Thus, entry 3 of our word-to-symbol map (which better have 8 unique entries in this example) will be the symbol representing the bit stream 110, say 4+4i. This complex amplitude is then stored in the output stream of parallel words, eventually to occupy an FFT bin.

The only potential pitfall here is the use of indexing. When an array is fed into a for loop, you can right click the entry point and enable indexing. This will present inside the loop the element corresponding to that iteration of the loop. In other words, the first time around, we're going to access the first bit-word (0th element), the second time around the second word etc. In that vein, we also build our output array in the same iterative fashion by indexing it as well.

If you are still confused on how this works, check out the tutorial video below, the example usage video in Figure 4 or email the author for more questions. The sub-VI is available below for download.

Instructional Video

[missing_resource: http://www.youtube.com/v/ScB9H3OQpWA&hl=en&fs=1&rel=0]

This is the instructional video for constructing the Words-to-Symbols sub-VI.

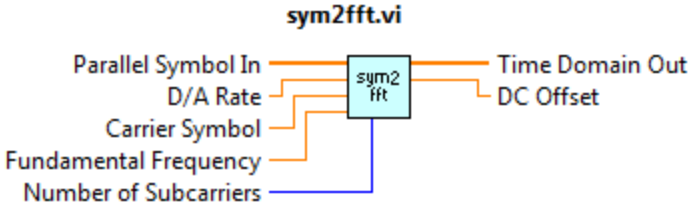
Example Video

[missing_resource: http://www.youtube.com/v/l_pRPWRVvy0&hl=en&fs=1&rel=0]

This is the example video for using the Words-to-Symbols sub-VI.

Download This LabVIEW sub-VI

Symbols-to-FFT (Transmitter) (LabVIEW)
This module is a tutorial on performing the parallel symbols to IFFT (time domain) operation of the OFDM transmitter. Often the most confusing and least intuitive block in the OFDM chain, a thorough understanding of the digital Fourier domain is recommended prior to tackling this function.
Input/Outputs and Help



This module accepts an array of symbols and performs the inverse fast Fourier transform (IFFT) to produce a time-domain signal. Essentially, it acts as a parallel-to-serial converter for the OFDM chain.

The carrier symbol is either injected in the case of an even number of subcarriers, or omitted in the case of an odd number of subcarriers. The spectrum is zero-padded such that the fundamental frequency and the rest of the subcarriers are properly spaced with respect to the Nyquist rate. Therefore, the number of samples will vary depending on the D/A rate and the fundamental frequency.

This is the LabVIEW help and block description for the Symbols-to-FFT Sub-VI.

This function (shown above in Figure 1) implements the most important and often the most confusing part of the OFDM transmitter chain: converting the parallel group of symbols to a continuous time-domain signal. Note that this module requires the Digital-to-Analog coverter Rate (D/A Rate) and the fundamental frequency of the subcarriers. Both of these parameters were not needed until now, because we've been discretely preparing our OFDM data.

The most frequently asked question about OFDM is "Why IFFT?" To those unfamiliar, IFFT stands for "Inverse Fast Fourier Transform." This is simply an efficient algorithm for performing the Inverse Discrete Fourier Transform, the sampled version of the IDTFT (Inverse Discrete Time Fourier Transform) which is the Fourier Transform for discretized signals.

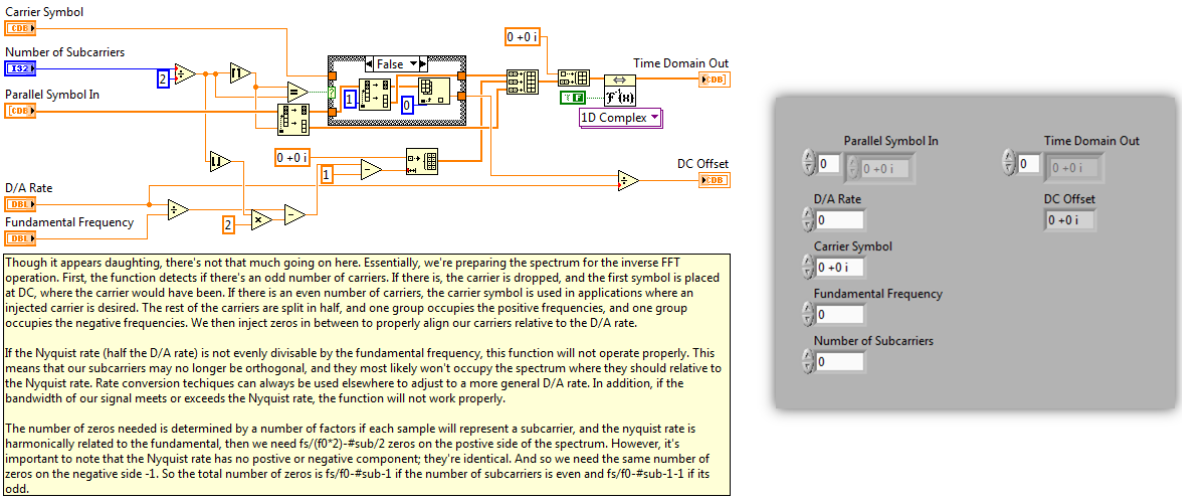
Confused yet? Don't worry. Simply understand the IFFT's function is to take the frequency-domain representation of a signal, and produce the time-domain equivalent.

So again you ask, why are we performing this step? Aren't we already in the time domain? Well, it's all rather relative. We know for OFDM the 'O' stands for Orthogonality. This attractive feature of OFDM signals makes the subcarriers insensitive to spectral overlap, much like Quadrature Multiplexing where we mix signals with cosine and sine of the same frequency. Thus as long as we modulate a variety of carriers all orthogonal to one another, we can neglect overlap of their spectra and still always recover each carrier separately without distortion. The orthogonality for OFDM comes from finding a fundamental frequency (lowest), and mixing all the other carriers with multiples of this fundamental, or harmonics. So how do we implement this?

Okay, let's look at a 32-subcarrier OFDM signal. We could modulate 32 frequencies with our 32 symbols and add them up, but this is extremely inefficient. Instead, we can start in the frequency domain, and place our symbols in adjacent samples. By doing this, because the samples are periodic, we're guarunteed to have all the frequency samples orthogonal to the first non-DC frequency. Then, since we can use Quadrature Multiplexing later on, we can simply place 15 symbols in the first 15 samples after DC, insert a bunch of zeros, and then place the last 15 symbols in the last 15 spots. So we now have 32 subcarriers modulated: +-fo, our fundamnetal, +-2fo, ... , +-14fo in the frequency domain. Essentially what we've done is build our signal from the spectrum. Because we have done this without any kind of conjugate symettry (at least not planned any), we will get a complex time-domain signal when the inverse FFT is taken. This is okay! We eventually will modulate cosine at our intermediate frequency with the real part of our baseband signal, and the imaginary part with sine.

Another way of looking at this is that we're simpling doing a parallel to serial operation. Let's get on to the actual implementation in LabVIEW.

Block Diagram Layout



This is the LabVIEW block diagram for the Symbols-to-FFT sub-VI.

Above we see the actual block diagram. Though it can appear overwhelming, as long as the theory discussed above is understood, the rest is rather simple. In fact, everything we discussed above is nearly implemented in one block by the built in IFFT function! Everything we see surrounding it is put in place for user-friendliness and flexability.

In case an injected carrier for ease of receiving is desired, the user can specify a carrier symbol. The array size divided by two and rounding simply is to find out if there is an even or odd number of subcarriers. If even, the carrier specified is injected. If odd, the DC frequency is injected with the first symbol, so the lone symbol will take the place of the carrier for the sake of symettry. The reason the round toward infinity is used is simply because of the way the split array function works. The split array function simply truncates the number and cuts the array at that point, so by rounding up the odd carrier will always be in the first half.

The second half of the split is added to the end of the frequency domain. The only magic then appears in between: how many zeros do we inject and why? The answer isn't as direct, but it is a very fundamental concept. We want to conserve bandwidth, and so this is the motivation between bunching up the subcarriers as close to DC as possible. In addition, the

easiest and most efficient way to make them orthogonal is to place a symbol at each sample. This way they're all related to the first non-DC frequency which is itself a fraction of the sampling (D/A) frequency.

Now, we want to ensure these relationships are maintained, as well as a few more. The user can specify the exact fundamental frequency desired, as well as the ultimate D/A rate. Two things must hold in the spectrum then: the fundamental frequency must be related to the D/A rate appropriately and in the first sample, and the Nyquist Rate of the system must be half the sampling frequency. If any of this is completely new, review the fundamentals of Digital Signal Processing (DSP). This constraint is the reason we choose the number of zeros, and why we want to choose our fundamental frequency such that sampling_freq/(2*fundamental_freq) is a whole number. Stated another way, the Nyquist Rate is evenly divisible by our fundamental. Once again, we can say the Nyquist Rate is an integer multiple of the Fundamental Frequency. This should be apparent from the above discussion. Afterall, halfway through our samples we should encounter the Nyquist Rate, the highest representable digital frequency, half the sampling rate. Every sample before this will be a fraction of this frequency. If this frequency exists at Sample 32, then our fundamental should be 1/32th of the Nyquist rate, as we will place it on a sample directly (no interpolation).

While a more generalized procedure is certainly possible, it's simply not practical in the face of transparency and computational complexity. In addition, sample rate conversion is a topic in and of itself, and any of those advanced techniques can be applied to the signal externally before transmission.

If you are still confused on how this works, check out the instructional and example video below or email the author for more questions. The sub-VI is available below for download.

Instructional Video

[missing_resource: http://www.youtube.com/v/PFeAlHbhkX8&hl=en&fs=1&rel=0]

This is the instructional video for constructing the Symbols-to-FFT prefix sub-VI.

Example Video

[missing_resource: http://www.youtube.com/v/-TmALixLjF4&hl=en&fs=1&rel=0]

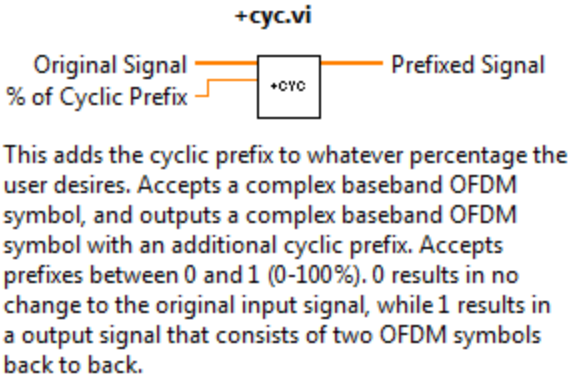
This is the example video for using the Symbols-to-FFT sub-VI.

Download This LabVIEW sub-VI

Add Cyclic Prefix (Transmitter) (LabVIEW)

This module discusses the cyclic prefix in the OFDM transmitter, both why it is needed and how it can be constructed in a LabVIEW environment. It contains the block diagram, instructional video, and file available for download.

Input/Outputs and Help



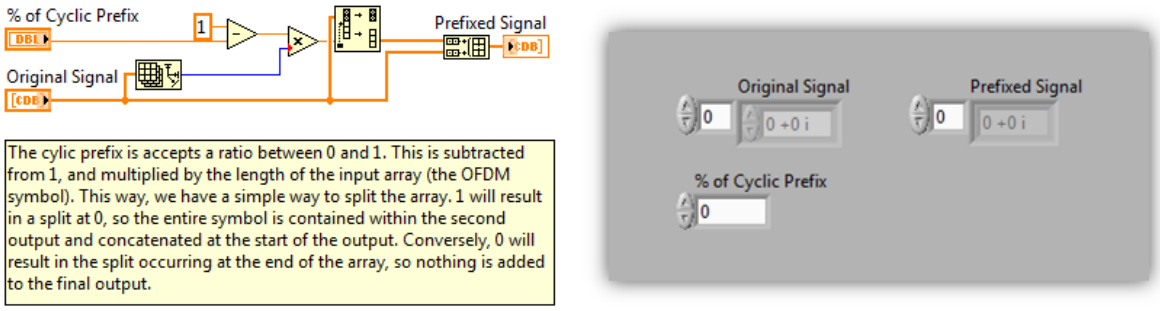
This is the LabVIEW help and block description for the cyclic prefix sub-VI.

The Cyclic Prefix sub-VI in the OFDM transmitter is shown above in Figure 1. The theory behind the cyclic prefix is rather intuitive. Leading up to this module, we have one complete period of a complex baseband OFDM symbol. However, depending on the channel, we may have significant multipath fading. In order to combat channels with delays, we need a guard interval. This is essentially an interval in the transmission that we will eventually throw out. For more complete treatment on multipath channel models and why guard intervals are effective, Krishna Sankar does a terrific job in his blog entry.

The procedure is very simple. Some percentage of the end of the signal is copied and added to the front. This doesn't create any discontinuities because the next point after the end of the signal is the beginning of the signal since we have exactly one period. Thus any portion of the end of the signal we copy to the front of the signal will add without discontinuity. If

this is still difficult to see, imagine copying and pasting a full period of a sine wave to the start, so we now have two periods of that sine wave. Now imagine deleting 1/4 of the points at the beginning of the new signal. It's easy to see that nothing unorthodox is going on here, and you've effectively created a cyclic prefix of 0.5 or 50%. Now it's easy to generalize this to any arbitrary signal, as long as we have exactly one period.

Block Diagram Layout



This is the LabVIEW block diagram for the cyclic prefix sub-VI.

Here, Figure 2 shows a LabVIEW implementation of what we just described. It is available for download below. The only subtleties here mainly deal with user friendly-ness. The module accepts a ratio between 0 and 1 representing a percentage between 0 and 100%. We then take the complement of this ratio in order to properly split the array. So our ratio is created by 1-(perc/100). This is because LabVIEW's built in function split array takes as an input the point in the array at which we wish to split. Thus if we want 75% prefix, since we work from the end of the signal and move towards the start, we want to break at the 25% point of the signal and take the second half. This ratio is thus multiplied by the total signal length and the break point is calculated.

The portion that is extracted is then concatenated to the original input signal, and our prefixed signal is complete. One sticking point is to ensure that when the "Build Array" function is invoked, that you right-click the block and choose "Concatenate Inputs" so that the arrays are properly joined. For all general questions, check out the instructional video below in

Figure 3, the example use video in Figure 4, or email the author for more information.

Instructional Video

[missing_resource: http://www.youtube.com/v/Zykhdaw_YkI&hl=en&fs=1&rel=0]

This is the instructional video for constructing the cyclic prefix sub-VI.

Example Video

[missing_resource: http://www.youtube.com/v/BzN5nWRsslA&hl=en&fs=1&rel=0]

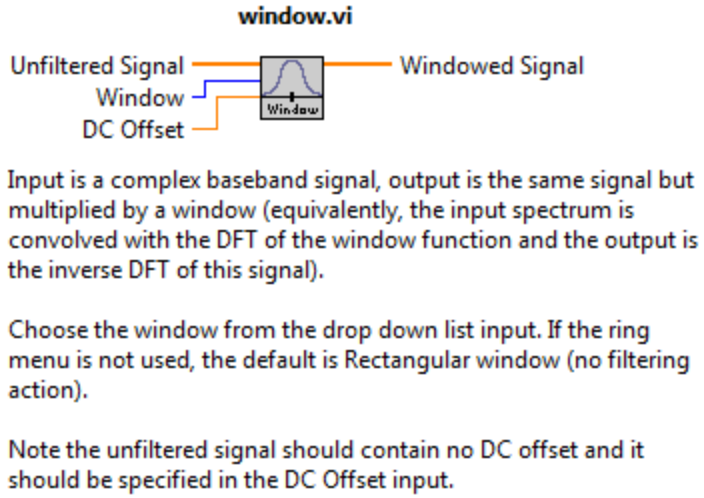
This is the example video for using the cyclic prefix sub-VI.

Download This LabVIEW sub-VI

Windowing (Transmitter) (LabVIEW)

This module demonstrates a LabVIEW implementation of a window function in an OFDM system. It also demonstrates why windowing is so critical in this particular case.

Input/Outputs and Help

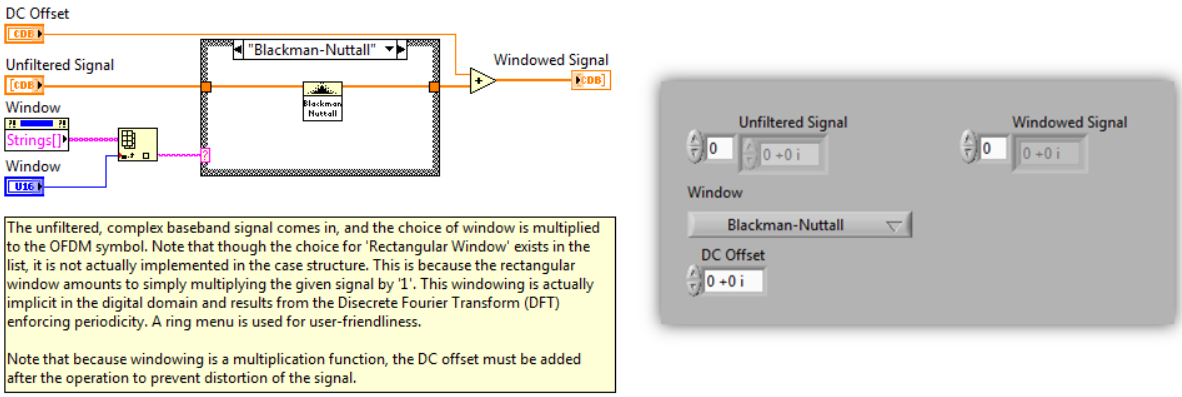


This is the LabVIEW help and block description for the Window Sub- VI.

The above figure shows the sub-VI snapshot of the rather simple module that windows the outgoing signal. This VI can be imperative in certain systems. The reason it becomes critical is the inevitable discontinuity that will arise between OFDM symbols.

If this is unclear, make sure you understand just how OFDM symbols are synthesized. The signals are built from symbol placement in the frequency domain at harmonics of the fundamental. As a result, there is no time-domain cohesiveness between symbols (or at least it occurs rarely). This is the same property (essentially) that plagues OFDM systems in the form of peak-to-average power ratios.

Block Diagram Layout



This is the LabVIEW block diagram for the Window sub-VI.

Pictured above is the LabVIEW block diagram implementation of the windowing process. Thankfully, almost all prominent window functions are built into LabVIEW. The windowing process consists of a simple multiplication in the time-domain. Those familiar with Fourier decomposition will recognize this corresponds to convolving in the frequency domain. Usually, practical windowing involves emphasizing the middle of the time domain signal, and deemphasizing the beginning and the end of the signal.

The reason for this deemphasis stems from the fundamental (but covert) function the Discrete Fourier Transform (DFT) performs. While subtle, it's important to note that by taking the DFT of a signal, one is enforcing periodicty of that signal. That is, we are transforming the signal into the frequency domain under the assumption that the input repeats in the time-domain indefinitely. As a result, the un-windowed signal (or equivalently, rectangularly windowed signals) present serious flaws from this poor assumption. This is especially manifested in our situation in which discontinuities are almost guarunteed to occur. Therefore, deemphasizing the end points, and consequently the discontinuities, proves fruitful.

The only slight of hand in the LabVIEW implementation is the window selection method. After creating a case structure, simply naming the various windows with strings and inserting them suffices. However, in the interest

of user-friendliness, an effective interface is needed. By creating a menu-ring, the options easily present themselves, and a simple array index operation allows us to access the case structure with ease. For more information, check out the instructional tutorial below, the example video in Figure 4, or email the author.

Instructional Video

[missing_resource: http://www.youtube.com/v/BDdNJam9xOU&hl=en&fs=1&rel=0]

This is the instructional video for constructing the windowing sub-VI.

Example Video

[missing_resource: http://www.youtube.com/v/QfGp69bniVo&hl=en&fs=1&rel=0]

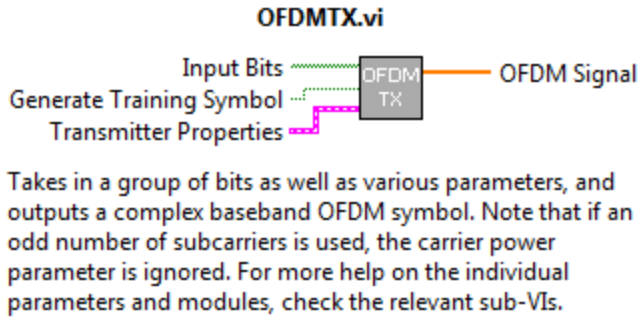
This is the example video for using the windowing sub-VI.

Download This LabVIEW sub-VI

OFDM Symbol Generator (Transmitter) (LabVIEW)

This is the culmination of all the previously developed sub-VIs. It's job is to simply package up all the functionality nice and neat, and make it ready to place in any communications template for real-world use.

Input/Outputs and Help

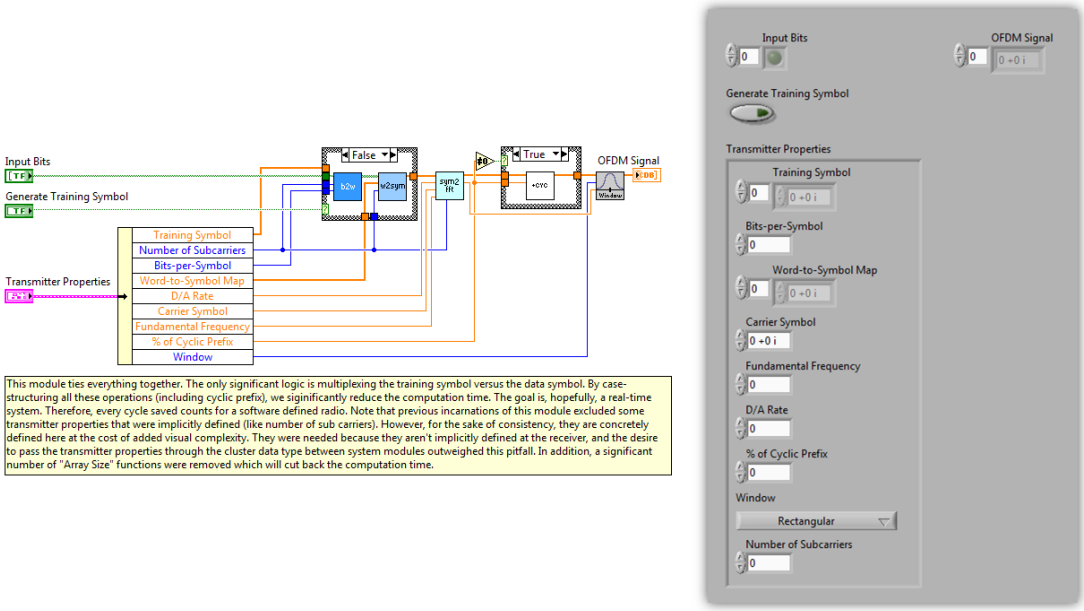


This is the LabVIEW help and block description for the OFDM Symbol Generator Sub-VI.

Putting together all of the previously built sub-VI's into a sub-VI itself, we get a convienent block (shown above in Figure 1) to drop into any software communications template in order to complete the system. By specifying all the input data, this block will output a signal OFDM time-domain signal ready to be quadrature multiplexed out to IF and actually transmitted.

Again, though some may find this module unecessary (and pedantically speaking they would be correct), modularity and transparency goes a long way in system integration. Almost always, the benefits outweigh the minimal time needed to develop such modules.

Block Diagram Layout



This is the LabVIEW block diagram for the OFDM Symbol Generator sub-VI.

There is little to discuss here, and the really only sticking point is the use of the case structures and clusters. One could have the blocks outside the case structures and feed their outputs through the case structures. However, by placing the functions in the case structure, we decide if we actually call the function or not, instead of using the output signal. This way we reduce unneeded computational time. The cluster data type is used for visual clarity, and ease of transferring transmitter properties to other systems (like the receiver).

If you are still confused on how any of this works, check out the instructional video below, the all software test module, or email the author for more questions. The sub-VI (as well as all VIs it uses) is available below for download.

Instructional Video

[missing_resource: http://www.youtube.com/v/QWzYEy9gshs&hl=en&fs=1&rel=0]

This is the instructional video for constructing the OFDM Symbol Generator sub-VI.

Download This LabVIEW sub-VI

NOTE: This zip file also contains all the sub-VI's shown in this module that this sub-VI uses.

Time Synchronizaton (Receiver) (MATLAB)

This demonstrates how to recover the symbol timing in an OFDM communications system. It relies on a training symbol and a timing metric in order to find the end of the cyclic prefix and the start of the OFDM frame. This module code will soon be supplanted by a more modern algorithm. There is an inherit plateau of uncertainty in the symbol timing with cyclic prefix, a necessity in wireless applications. More modern algorithms and training symbols more effectively reject the cyclic prefix and give a more accurate location of the start of the frame. **NOTE: The MATLAB library was used in the initial version of the design but has since been replaced by an updated LabVIEW module. Please see the LabVIEW portion of the receiver for the most current version.**

```
%% Time Synchronization (RECEIVER) % ------------
-------------------------------------------------
--------- % Description: This implements Schmidl
and Cox's symbol frame timing % synchronization
algorithm. The idea is that by zeroing out % the
odd frequencies for the training symbol, we
produce an % OFDM symbol that has two periods
within the normal OFDM % symbol frame, whereas a
normal symbol has only one period. % By sliding a
window in time and multiplying sample by % sample
two successive windows of length half of the
normal % symbol and one of them conjugated, the
magnitude peak % should hit '1' and represent the
exact start of the frame % for the training
symbol. Thus it as well as the data symbol % can
be extracted % % Inputs: signal - Input waveform,
sampled from LABVIEW % size_of_fft - Size of fft
for symbol size % Outputs: index - Index of start
of timing frame % agc - Automatic gain control
factor function [index agc] = tsync(signal,
size_of_fft) %% signal_size = size(signal,2); %
Signal size L = size_of_fft/2; % Length of sliding
window slide_length = signal_size-size_of_fft+1; %
```

```
Total window slide length P =
zeros(1,slide_length); % Initilize the arrays R =
zeros(1,slide_length); for n=1:slide_length for
m=1:L P(n) = P(n) + conj(signal(n+m-
1))*signal(n+m+L-1); % Conjugate pair R(n) = R(n)
+ abs(signal(n+m+L-1))^2; % Normalizing factor end
end M = (abs(P).^2);%./(R.^2); % This is the
actual timing metric used. Note the algorithm
calls for % the normalization factor. However, due
to peak-to-average power % issues, we experienced
issues with the algorithm choosing false peaks %
that were greater than 1. After much
experimenting, removing the % normalization factor
and boosting the signal power of just the % odd-
channels in the training symbols made this
algorithm robust. for n=1:slide_length [value
index] = max(M); % Find peak
if((index+2*size_of_fft-1) > signal_size) M =
M(1:index-1); % If peak is near the end, and the
symbols cut off, back up and pick another one.
else break end end agc = R(index); % Output
normalizing factor for automatic gain control
(optional) end
```

Frequency Synchronization (Receiver) (MATLAB)

When an OFDM symbol is received, there is almost a guarantee the mixing frequency will be out of phase and slightly different than the carrier frequency. This algorithm is designed to eliminate this inconsistency by way of a predetermined training symbol. This module code will soon be supplanted by a more modern algorithm. For BPSK, we can model the frequency offset as a constant phase offset and the algorithm works fine. For more general forms of Quadrature Amplitude Modulation (QAM), this does not suffice. **NOTE: The MATLAB library was used in the initial version of the design but has since been replaced by an updated LabVIEW module. Please see the LabVIEW portion of the receiver for the most current version.**

```
%% Frequency Synchronization (RECEIVER) % --------
---------------------------------------------------
-------------- % Description: This module attempts
to correct the frequency/phase offset % in the
recovered signal by estimating it as a static
phase % offset. Though there may be a frequency
offset or even a % frequency drift, for BPSK,
estimating this as a fixed phase % offset is
sufficient in recovering the bits. More often %
than not this corrects the recovered bits when
they're % inverted (estimated as a pi phase
offset). This uses the % same metric as the timing
recovery to find the phase. % % Inputs: train -
Received training symbol % train_expected -
Expected training symbol via a priori knowledge %
Outputs: phi - Estimated phase offset function phi
= fsync(train, train_expected) %% L =
size(train,2); % Size of window for algorithm P =
0; % Initialize the metric for n=1:L P = P +
conj(train(n))*train_expected(n); % Sample by
sample multiplication of the complex conjugate of
the % received training symbol with the expected
training symbol. If % both of them are in phase,
```

```
the imaginary components would % annhiliate each other. However, if there's a phase offset % between the two, it will be represented in the angle of the % resulting product. end phi = angle(P); end
```

FFT-to-Symbols (Receiver) (MATLAB)

The incoming OFDM symbol was generated by essentially performing a parallel to serial transformation via an Inverse Fourier Transform (IFT). This is frequently done in the digital domain by using the Inverse Fast Fourier Transform (FFT) algorithm. Thus, we wish to recover our spectrum where the bits were present in parallel. For more information on why we use IFFT in the chain of the OFDM transmitter, see the appropriate module.

NOTE: The MATLAB library was used in the initial version of the design but has since been replaced by an updated LabVIEW module. Please see the LabVIEW portion of the receiver for the most current version.

```
%% FFT to Symbols (RECEIVER) % ------------------
------------------------------------------------
--- % Description: Takes a received OFDM signal
that has been demodulated to % baseband and
corrected for frequency/phase offset, and %
extracts the complex envelope of each carrier.
This % converts the time domain signal into a
descrete set of % symbols that can be mapped to
binary words. % % Inputs: re - Real channel of
OFDM symbol % im - Imaginary channel of OFDM
symbol % num_subcarriers - Number of subcarriers
used in OFDM symbol % Outputs: symbols - Extracted
complex envelopes of each subcarrier function
symbols = fft2sym(re, im, num_subcarriers) %%
size_of_fft = size(re,2); % Implicit size of fft
output = fft(re+j*im); % Time domain signal was
generated using ifft() symbols = [
output(2:1+num_subcarriers/2)
output(size_of_fft+1-
num_subcarriers/2:size_of_fft) ]; % Save only the
FFT samples that contain complex symbols as
specified % by the number of subcarriers. end
```

Symbols-to-Words (Receiver) (MATLAB)
This is a basic building block in most nontrivial communications schemes: mapping received symbols to bit words. For more thorough treatment on this concept, read up on "Bit-to-Symbol Mapping." **NOTE: The MATLAB library was used in the initial version of the design but has since been replaced by an updated LabVIEW module. Please see the LabVIEW portion of the receiver for the most current version.**

```
%% Symbols to Words (RECEIVER) % ----------------
---------------------------------------------
----- % Description: Converts the received symbols
from the OFDM symbol spectrum % into binary words.
It does this by correlating the received % symbol
with the entire symbol map through a minimization
of % the absolute value of the difference with
each. Note that % if one uses multiple
constellation points in the same % quandrant of
the complex plane, it may be necessary to add %
additional logic here, or ensure the channel
effect is % removed by equalization. % % Inputs:
symbols - Symbols extracted from the OFDM symbol %
word_to_symbol_map - The complex symbols indexed
by the binary % word they represent. % Outputs:
words - Closest matched binary words after
comparison function words = sym2w(symbols,
word_to_symbol_map) %% num_subcarriers =
size(symbols,2); % Implicit number of sub-carriers
bits_per_symbol =
log2(size(word_to_symbol_map,2)); % Implicit
number of bits per symbol to represent each word
words = zeros(num_subcarriers, bits_per_symbol); %
Initialized array for n=1:num_subcarriers [v i] =
min(abs(word_to_symbol_map-symbols(n))); %
Subtract each symbol with every symbol in the map,
and find the % minimum absolute value. This is a
great estimate for finding the % received word.
```

```
word = dec2bin(i-1,bits_per_symbol); % Convert
index to binary string for m=1:bits_per_symbol
words(n,m) = str2num(word(m)); % Convert string to
number end end end
```

Words-to-Bits (Receiver) (MATLAB)
A very fundamental part of many systems, essentially a parallel to series conversion. **NOTE: The MATLAB library was used in the initial version of the design but has since been replaced by an updated LabVIEW module. Please see the LabVIEW portion of the receiver for the most current version.**

```
%% Words to Bits (RECEIVER) % ---------------------
-------------------------------------------------
-- % Description: This module simply takes the
demodulated words and expands % them into a bit
stream for ease of comparison. % % Inputs: words -
Group of bits % Outputs: bits - Bit stream
function bits = w2b(words) %% num_subcarriers =
size(words,1); % Implicit number of subcarriers
bits_per_symbol = size(words,2); % Implicit bits
per symbol total_bits =
bits_per_symbol*num_subcarriers; % Total number of
bits bits = zeros(1,total_bits); for
n=1:num_subcarriers for m=1:bits_per_symbol
bits((n-1)*bits_per_symbol+m) = words(n,m); %
Separate one by one end end end
```

OFDM Symbol Decoder (Receiver) (MATLAB)
This file brings together all of the previous files into one MATLAB interface. With one command, and the right parameters, one can successfully receive one training symbol and one data symbol. **NOTE: The MATLAB library was used in the initial version of the design but has since been replaced by an updated LabVIEW module. Please see the LabVIEW portion of the receiver for the most current version.**

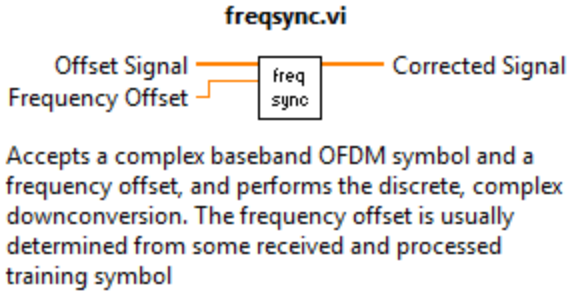
```
%% Demodulate Baseband OFDM Signal (RECEIVER) % --
---------------------------------------------
-------------------- % Description: This is the
receiver module in its entirity. It combines % all
of the sub-processes and parameters to ultimately
take % a sequency of training symbol and valid
OFDM symbol, and % output the modulated bits. % %
Inputs: word_to_symbol_map - The complex symbols
indexed by the binary % word they represent. %
num_subcarriers - The number of subcarriers in the
symbol % size_of_fft - Size of FFT used for
correct indexing/timing % train_pn - Pseudo-noise
used in transmitted training symbol (a % priori
knowledge) % Outputs: bits - Demodulated bits
function bits = ofdm_dem(word_to_symbol_map,
num_subcarriers, size_of_fft, train_pn) %% %
Calculated the expected training symbol from the
given pseudo-noise for % adequate phase recovery.
train_expected = zeros(1,size_of_fft);
train_expected(3) = train_pn(1);
train_expected(size_of_fft-1) =
train_pn(1+size(train_pn,2)/2); train_expected =
ifft(4*train_expected); z = get_input(); % Parse
input data z = z'; % Convert from column vector to
row vector to accomidate sub-processes % Invoke
timing algorithm to retrieve sample that training
symbol starts on [index value] =
tsync(z,size_of_fft); train =
```

```
z(index:index+size_of_fft-1); % Cut out training symbol phi = fsync(train, train_expected); % Estimate phase offset % Cut out OFDM data symbol and correct for phase offset z = z(index+size_of_fft:index+2*size_of_fft-1).*exp(j*phi); % Separate the data symbol into two channels and take FFT for analysis % below re = real(z); im = imag(z); Z = fft(z); symbols = fft2sym(re,im,num_subcarriers); % Take FFT and retrieve symbols words = sym2w(symbols,word_to_symbol_map); % Find closest match and map words to symbols bits = fliplr(w2b(words)); % Decompress words into a series of bits for recovery. % Output the resulting data for debugging and analysis subplot(5,2,1); plot(real(train)); title('Re[Training Symbol]'); subplot(5,2,2); plot(imag(train)); title('Im[Training Symbol]'); subplot(5,2,3); plot(re); title('Re[OFDM Data Symbol]'); subplot(5,2,4); plot(im); title('Im[OFDM Data Symbol]'); subplot(5,2,[5 6]); plot(abs(Z)); title('Mag[OFDM Data Symbol Spectrum]'); subplot(5,2,[7 8]); stem(real(Z)); title('Re[OFDM Data Symbol Spectrum]'); subplot(5,2,[9 10]); stem(imag(Z)); title('Im[OFDM Data Symbol Spectrum]'); end
```

Frequency Synchronization (Receiver) (LabVIEW)

In this module we implement a simple complex downconversion based on a predetermined frequency offset. The reason for complex (by way of complex exponential multiplication) is because we expect (in general) no conjugate symmetry for our incoming signal. Embedded is a tutorial video on the construction of the sub-VI, as well as an example video on how to use the function. The file is available for download below.

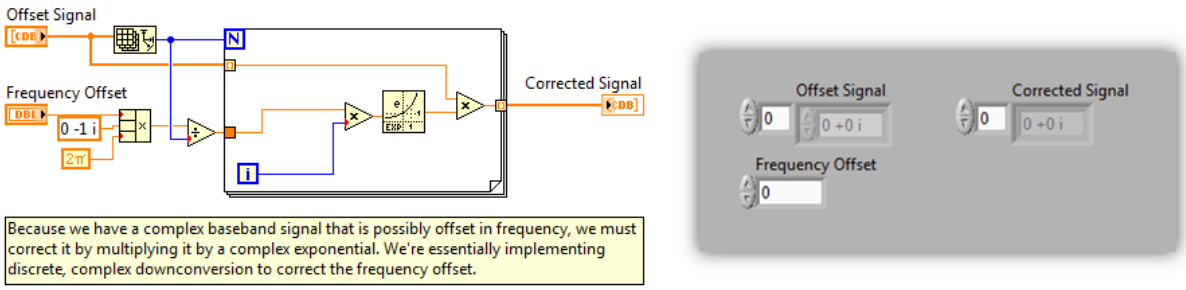
Input/Outputs and Help



This is the LabVIEW help and block description for the Frequency Synchronization sub-VI.

The Frequency Synchronization sub-VI in the OFDM transmitter is shown above in Figure 1. The motivation behind this sub-VI arises because we assume in general we have a complex baseband OFDM symbol. As a result, any frequency offset from noncoherence up until this point must be corrected by a complex downconversion. Of course, maintaining coherence is crucial in preserving the orthogonality of the subcarriers and ultimately in distortionless recovery of the data.

Block Diagram Layout



This is the LabVIEW block diagram for the Frequency Synchronization sub-VI.

Figure 2 shows a LabVIEW implementation of this downconversion and is available for download below. The only potentially unseen function for the new LabVIEW user is the compound arithmetic function. This can be found in the numeric menu. Once placed in your VI, you must right-click the block and choose the arithmetic operation you desire (multiplication in our case). Note also that indexing is enabled for ease of implementation.

For all general questions, check out the instructional video below in Figure 3, the example use video below in Figure 4, or email the author for more information.

Instructional Video

[missing_resource: http://www.youtube.com/v/FBN7BC6CA0A&hl=en&fs=1&rel=0]

This is the instructional video for constructing the Frequency Synchronization sub-VI.

Example Video

[missing_resource: http://www.youtube.com/v/haxRSUTu9Gg&hl=en&fs=1&rel=0]

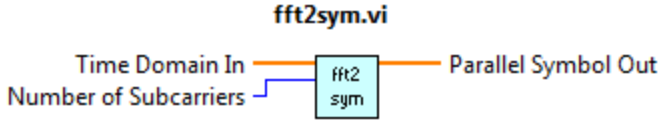
This is the example video for using the Frequency Synchronization sub-VI.

Download This LabVIEW sub-VI

FFT-to-Symbols (Receiver) (LabVIEW)

Once we've obtained our complex baseband data we need to recover the subcarriers. Because they were created using the Inverse Fast Fourier Transform (IFFT), we simply need to reverse the operation by taking the FFT. Once this is done, we will generally have a certain number of transmitted zeros between the positive and negative frequency bins that need to be removed to output our received data symbols.

Input/Outputs and Help

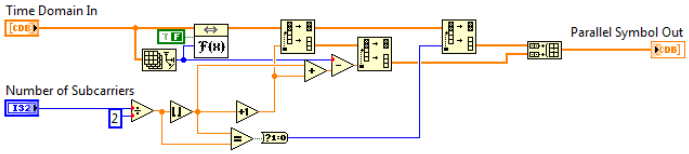


Takes the time domain received OFDM symbol and the number of subcarriers, and outputs only the subcarrier symbols. Note that the signal must have gone through the appropriate rate conversions to bring it to the state it was in when it was generated using the sym2fft sub-VI. This ensures there is precisely one subcarrier per FFT bin.

This is the LabVIEW help and block description for the FFT-to-Symbols sub-VI.

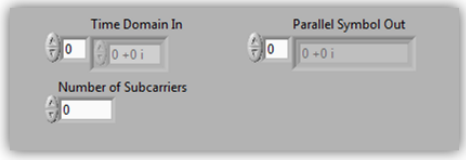
Shown above is the sub-VI to be discussed next. Once we receive our time domain signal, we need to recover the subcarriers and ultimately the data they contain. We took the IFFT at the transmitter, and so it is a simple matter of taking the FFT and removing the padded zeros.

Block Diagram Layout



This takes the incoming baseband OFDM symbol and extracts the subcarriers while discarding what was transmitted as zeros and has no doubt become noise and spurious emissions. The only switching operation determines if the DC offset is an injected carrier to be discarded or an odd subcarrier to be kept.

The only complexity stems from where to break up the array. After dividing the number of subcarriers, rounding down, and adding 1, we have accounted for the first block regardless if their is an even or odd number of subcarriers. After that, we subtract the first total plus the half the number of subcarriers to obtain the number of zeros that need to be cut and split the array appropriately.



This is the LabVIEW block diagram for the FFT-to-Symbols sub-VI.

Figure 2 shows the block diagram of our implementation available for download below. After taking the FFT, we simply determine how much of the spectrum should be data, and remove it from the rest of the array. While we transmitted a specific number of zeros in between the positive and negative frequencies, they will no doubt take on some small non-zero value after realistic travel through a non-software medium.

The only real logic is in dividing the number of subcarriers by 2 to determine if we have an even or odd number of subcarriers. If even, we can throw away the first data point as it is simply an injected carrier (or nothing). However, if odd, it instead contains a data point for the odd subcarrier and must be preserved in the output. For additional information, check out the tutorial video in Figure 3, the example video in Figure 4, or email the author with any questions.

Instructional Video

[missing_resource: http://www.youtube.com/v/6uZ5N1cR7CA&hl=en&fs=1&rel=0]

This is the instructional video for constructing the FFT-to-Symbols sub-VI.

Example Video

[missing_resource: http://www.youtube.com/v/1GJqHmOhk28&hl=en&fs=1&rel=0]

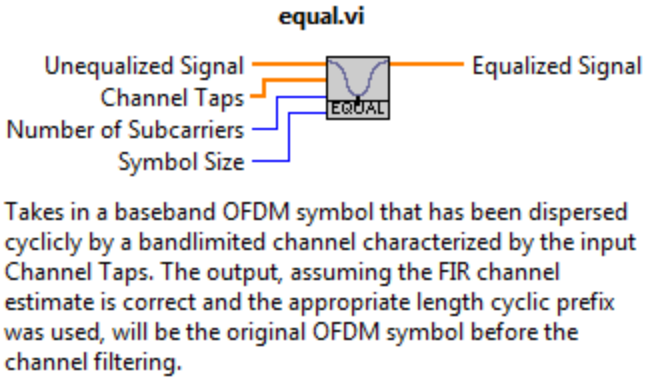
This is the example video for using the FFT-to-Symbols sub-VI.

Download This LabVIEW sub-VI

Equalization (Receiver) (LabVIEW)

After the data carriers are recovered, we know that if the appropriate cyclic prefix was used, our data has be cyclicly convolved with the channel impulse response. With an estimate of the channel taps, it becomes a simple division in the frequency domain to remove this undesired filtering. Embedded is a tutorial video on the construction of the sub-VI, as well as an example video on how to use the function. The file is available for download below.

Input/Outputs and Help

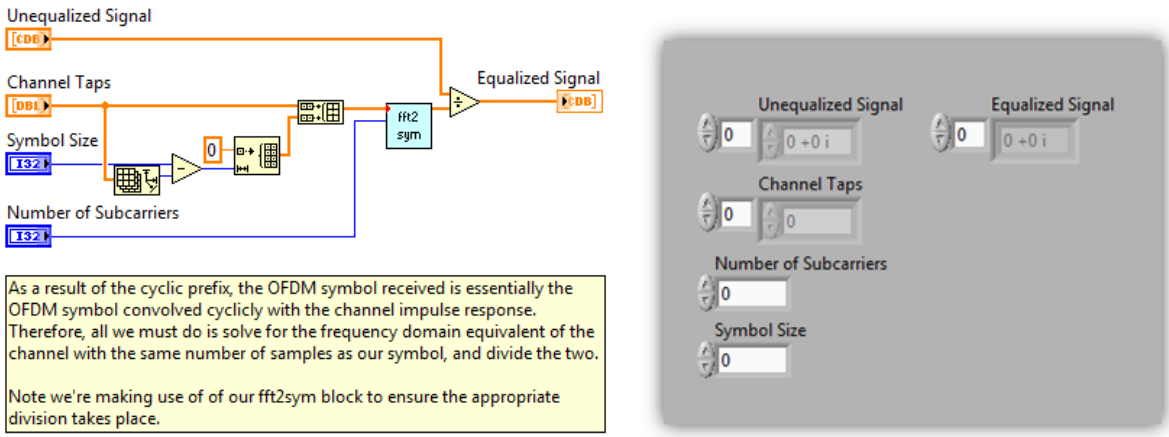


This is the LabVIEW help and block description for the Equalization sub-VI.

In Figure 1 we show the help dialog for the Equalization sub-VI to be implemented below. The motivation is that nearly every practical channel is bandlimited, and so our received signal will ultimately be the transmitted signal filtered by the dispersive channel. The resulting intersymbol interference (ISI) is the reason why a cyclic prefix, or guard interval, is required.

However, even if the cyclic prefix takes the brunt of the intersymbol interference, our received OFDM symbol has still experienced cyclic convolution with the channel impulse response. Thankfully, this amounts to normal division in the frequency domain if we can estimate the channel taps.

Block Diagram Layout



This is the LabVIEW block diagram for the Equalization sub-VI.

The implementation to be discussed in this module can be seen above in Figure 2. It's quite simple for those familiar with basic DSP. If we know the time domain impulse response of the channel (or can estimate it), we can simply take the FFT to obtain the frequency response. After removal of the cyclic prefix, it's as if the channel frequency response was simply multiplied by our transmitted OFDM symbol with no zero-padding for either digital spectrum. Therefore, all we must do is divide to undo this unwanted distortion.

Note that we have taken advantage of our fft2sym sub-VI implemented previously. For all general questions, check out the instructional video below in Figure 3, the example use video below in Figure 4, or email the author for more information.

Instructional Video

[missing_resource: http://www.youtube.com/v/rIRg2C5w31U&hl=en&fs=1&rel=0]

This is the instructional video for constructing the Equalization sub-VI.

Example Video

[missing_resource: http://www.youtube.com/v/-vtb-fBAWhQ&hl=en&fs=1&rel=0]

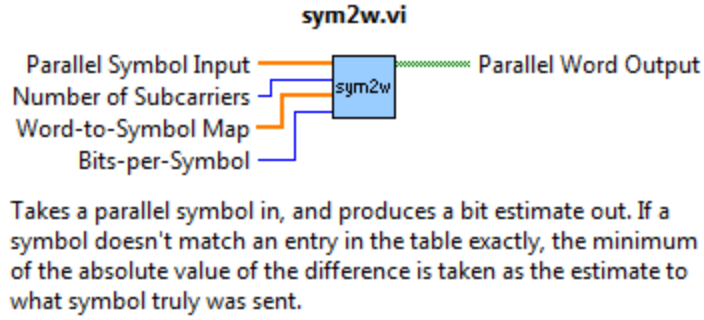
This is the example video for using the Equalization sub-VI.

Download This LabVIEW sub-VI

Symbols-to-Words (Receiver) (LabVIEW)

Once we've obtained our complex baseband data and recovered the subcarriers, it's time to make a decision. We have knowledge of what the potential values of the complex amplitudes are allowed to be, and we must fit what we receive to the closest value in the list. Therefore, an estimate of the transmitted symbol is ultimately the output. Embedded is a tutorial video on the construction of the sub-VI, as well as an example video on how to use the function. The file is available for download below.

Input/Outputs and Help

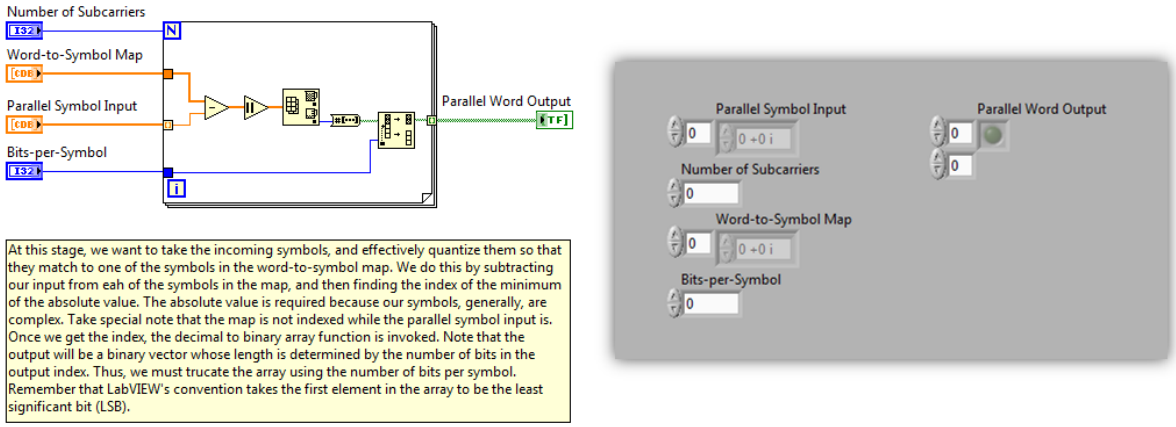


This is the LabVIEW help and block description for the Symbols-to-Words sub-VI.

Above is the sub-VI we intend to build in this module. This is the first module where we've attempted to address the stochastic nature of our received data. We'll take each received symbol and compare it with our word-to-symbol map. The closest match is determined to be the most likely transmitted symbol.

The reason the incoming signals may not perfectly match our transmitted symbols is because realistically, computational artifacts from the FFT operation, noise, dispersive channels, quantization, and imperfect coherent carrier recovery are just a handful of the potential sources that can cause our data to change from the transmitter to the receiver.

Block Diagram Layout



This is the LabVIEW block diagram for the Symbols-to-Words sub-VI.

Above, Figure 2 shows a LabVIEW implementation of our slicer. As alluded to, we essentially compute minimum Euclidean distance between all allowed values of our symbols (from our known map) and our received data one subcarrier at a time. It's important to note indexing is enabled for our incoming data to perform our operation one data carrier at a time, but the word-to-symbol map is not, as we want to compare each incoming data symbol with all symbols in the map.

Once we take advantage of LabVIEW's max/min function, we desire only the index as this corresponds to the decimal equivalent of our binary word. Remember LabVIEW's convention that the first element in the output array corresponds to the least significant bit. Because the number of bits varies depending on the size of the map, the truncation shown at the end is necessary to remove excess leading zeros. For all general questions, check out the instructional video below in Figure 3, the example use video below in Figure 4, or email the author for more information.

Instructional Video

[missing_resource: http://www.youtube.com/v/dpdp_u6lgOA&hl=en&fs=1&rel=0]

This is the instructional video for constructing the Symbols-to-Words sub-VI.

Example Video

[missing_resource: http://www.youtube.com/v/jSvw40Sp1C8&hl=en&fs=1&rel=0]

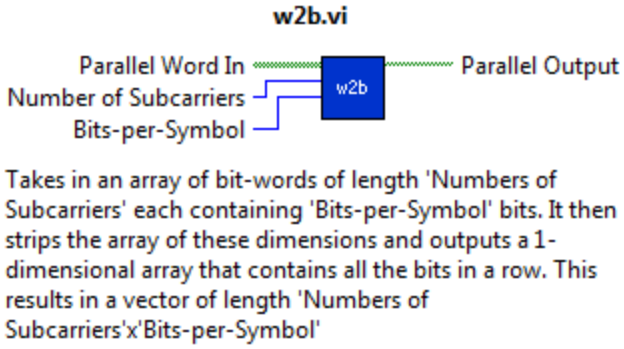
This is the example video for using the Symbols-to-Words sub-VI.

Download This LabVIEW sub-VI

Words-to-Bits (Receiver) (LabVIEW)

This is the instructional module for construction and use of the words-to-bits sub-VI for our OFDM receiver chain. Though somewhat trivial in nature, keeping it as it's own sub-VI helps maintain modularity should we wish to pursue more advanced implementations in the future. Embedded is a tutorial video on the construction of the sub-VI, as well as an example video on how to use the function. The file is available for download below.

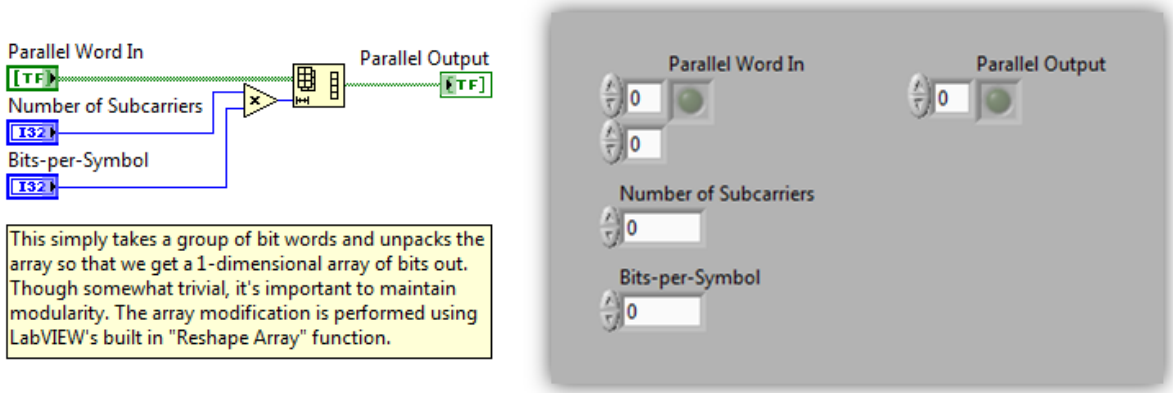
Input/Outputs and Help



This is the LabVIEW help and block description for the Word-to-Bits sub-VI.

We come again to a seemingly undeserving sub-VI shown above. While it may not warrant a sub-VI at present, future implementations will benefit from it's separation from other sub-VIs. This way, upgrading and changing individual parts will have minimal impact on the adjacent subsystems.

Block Diagram Layout



This is the LabVIEW block diagram for the Words-to-Bits sub-VI.

The above figure is our simple implementation as discussed above. All we're doing is taking advantage of LabVIEW's reshape array function. We're reshaping our two dimensional array defined by the input integers to one continuous array comprising one dimension.

Note that optimal implementations at lower levels could be achieved implicitly by how the memory is mapped. Therefore this module is simply a convenient educational tool for transparency in observing the overall OFDM transceiver hierarchy. For any inquiries, check out the instructional video below in Figure 3, the example use video below in Figure 4, or email the author for more information.

Instructional Video

[missing_resource: http://www.youtube.com/v/q4_1VWejcgI&hl=en&fs=1&rel=0]

This is the instructional video for constructing the Words-to-Bits sub-VI.

Example Video

[missing_resource: http://www.youtube.com/v/pxURvYHCK2A&hl=en&fs=1&rel=0]

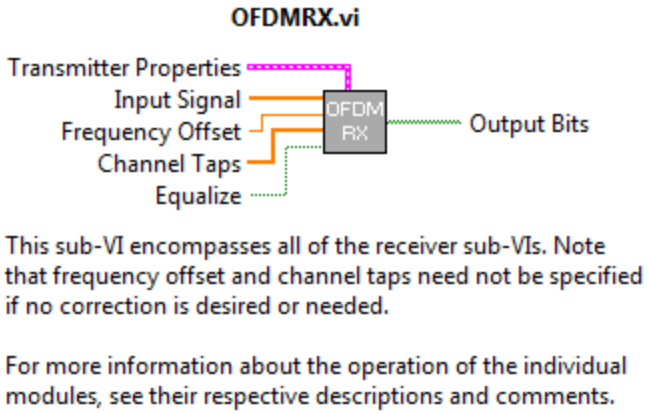
This is the example video for using the Words-to-Bits sub-VI.

Download This LabVIEW sub-VI

OFDM Symbol Decoder (Receiver) (LabVIEW)

This is the complete receiver package to close out our OFDM transceiver system. With this module, one can learn how to cascade the previously implemented sub-VIs to create a single entity capable of decoding OFDM symbols generated by the previously discussed OFDM symbol generator. Embedded is a tutorial video on the construction of the sub-VI, and the file is available for download below.

Input/Outputs and Help

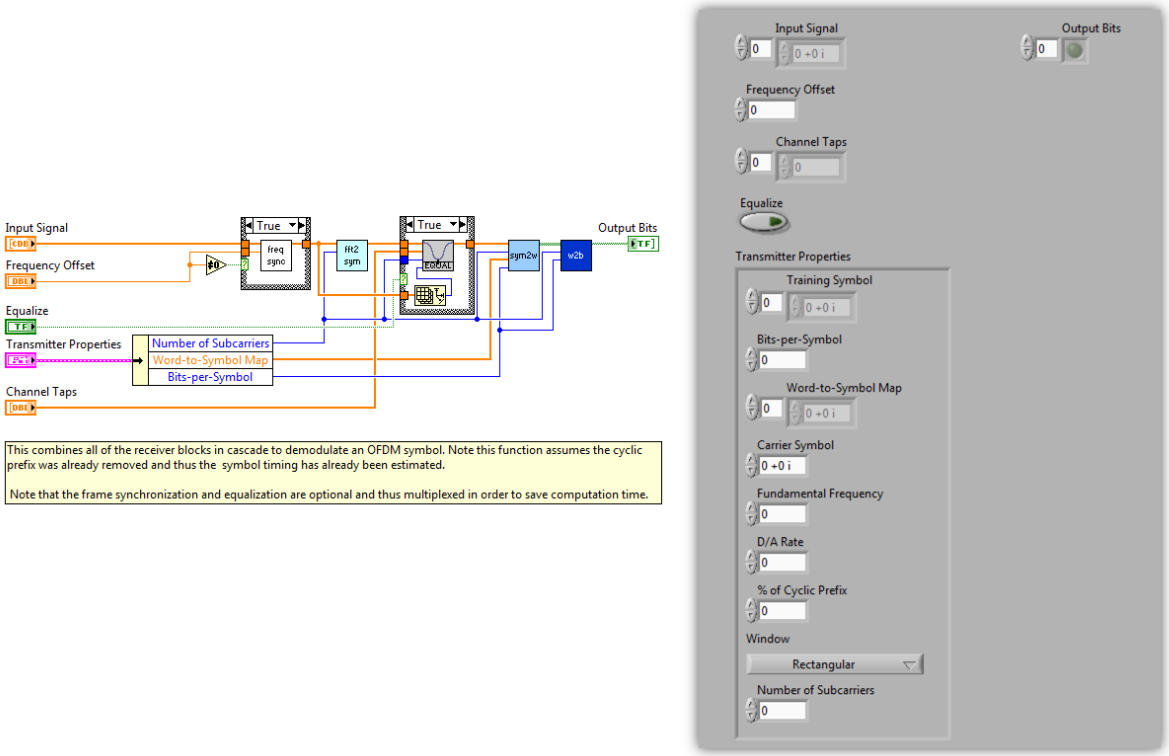


This is the LabVIEW help and block description for the OFDM Symbol Decoder sub-VI.

Show above is the LabVIEW sub-VI for receiving generic OFDM symbols to be placed in any real-life communications template. It is designed to be the polar-opposite of the OFDM symbol generator demonstrated previously, but can easily be placed in cascade in higher level VIs to tailor to any OFDM system.

Note that this module will be built from previously discussed receiver modules, so if anything is completely alien, check the menu list to the left for information about each sub-VI contained here.

Block Diagram Layout



This is the LabVIEW block diagram for the OFDM Symbol Decoder sub-VI.

As promised, here is our receiver built from all our previous sub-VIs. Note that, as with the OFDM symbol generator, multiple functions are multiplexed in case they're unneeded to save computational complexity.

The VI is built such that it can easily take a transmitter property cluster used in the symbol generation to configure the receiver blocks. Everything seen here has been previously seen prior, and the only pitfall is the use of clusters. We're now ready to test the complete OFDM chain in software to verify the performance of our transceiver. For more information, check the instructional video below and the all software test shown in the left hand column of the collection.

Instructional Video

[missing_resource: http://www.youtube.com/v/ZH4dXdYZzSs&hl=en&fs=1&rel=0]

This is the instructional video for constructing the OFDM Symbol Decoder sub-VI.

Download This LabVIEW sub-VI

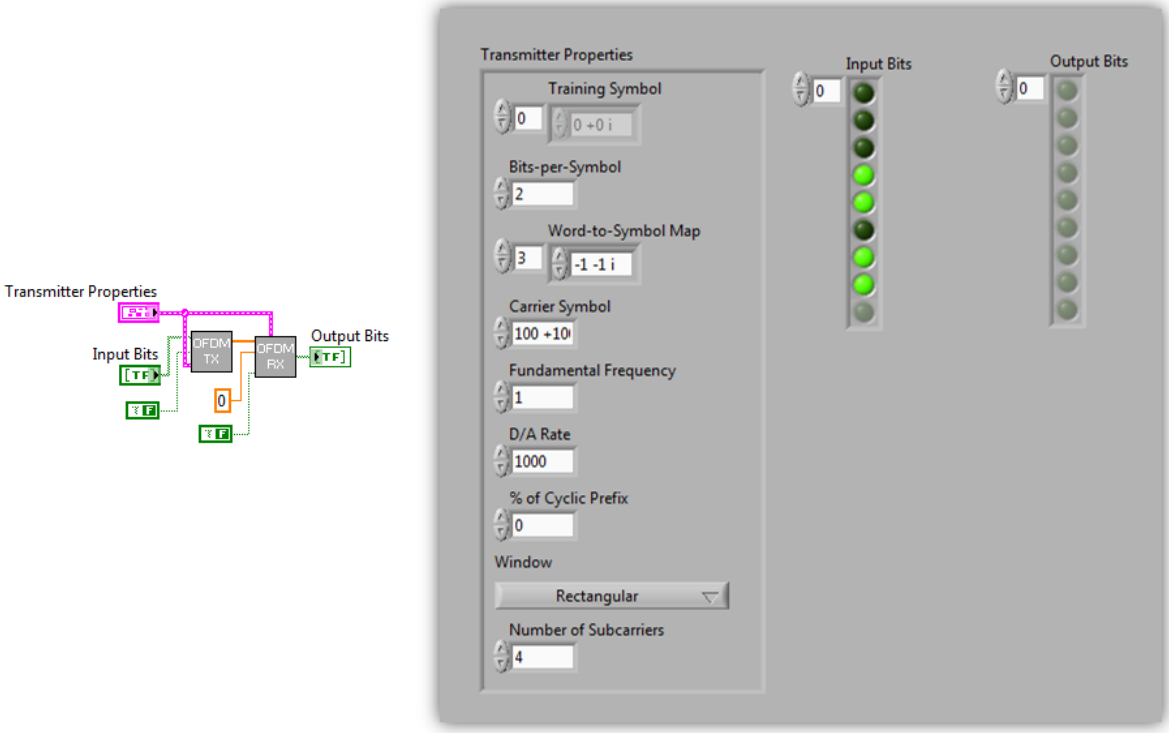
NOTE: This zip file also contains all the sub-VI's shown in this module that this sub-VI uses.

All Software (Tests)

This is the first round of testing on our complete system. It is simply the transmitter and receiver modules in cascade. The motivation is simply to confirm that our system works purely in software, output to input. The VI used is available for download, and the instructional and example video are embedded below.

In this module we're ready for a simple test. Before we can place either of our blocks into real-time communications systems, it's prudent to check that they work in an idealized situation. Thus we'd like to test them all in the HOST in LabVIEW

Block Diagram Layout



This is the LabVIEW block diagram for the All Software Test.

Above in Figure 1 is the setup for our test. Very straightforward. Check out the video below to see it in action, and download the VI at the bottom.

Example Video

[missing_resource: http://www.youtube.com/v/A4LxvNBZ71Y&hl=en&fs=1&rel=0]

This is the example video for using the OFDM Transmitter and Receiver blocks.

Download This LabVIEW sub-VI

References

This is an incomplete list of references used in the creation of this project. To acknowledge and trace every source of every bit of information contained here would be an intractable exercise in futility. That said, I want to thank everyone and anyone not explicitly mentioned in this list that had any hand in making this project a reality. I could not have done it without you. Thank you.

SPECIAL THANKS:

My beautiful wife Aimee, Dr. Christopher Schmitz, Dr. Douglas Jones, Aditya Jain, Steve Jian, Christopher Li.

REFERENCES:

[1] L. Couch, “Digital and Analog Communication Systems,” 2007.

[2] B. Farhang-Boroujeny, “Signal Processing Techniques for Software Radios,” 2008.

[3] C. Li, C. Schmitz, and A. Muehlfeld, “Building FPGA Communications Projects with LabVIEW,” Connexions. August 16, 2009. Available: http://cnx.org/content/m31349/latest. [Accessed: Oct. 16, 2009].

[4] T. Schmidl and D. Cox, “Robust Frequency and Timing Synchronization for OFDM,” IEEE Transactions on Communications, vol. 45, no. 12, pp. 1613-1621, Dec. 1997.

[5] P. Koch and R. Prasad, “The Universal Handset,” IEEE Spectrum, vol. 36, no. 4, pp. 36-41, Apr. 2009.

[6] C. Langton, “Orthogonal Frequency Division Multiplexing (OFDM) Tutorial,” Complex2Real.com: Intuitive Guide to Principles of Communications. 2004. Available: http://www.complextoreal.com/chapters/ofdm2.pdf. [Accessed: Oct. 16, 2009].

[7] H. Minn, M. Zeng, and V. K. Bhargava, “On Timing Offset Estimation for OFDM Systems,” IEEE Communications Letters, vol. 4, no. 7, pp. 242-244, Jul. 2000.

[8] C. Schmitz, “ECE 463: Digital Communications Laboratory,” ECE Illinois. Spring 2010. Available: http://courses.ece.illinois.edu/ece463/SP10/index.html. [Accessed: May. 1, 2010].